

CrackMe 6 – TTO

solution par mastermatt29

1.Introduction.....	2
2.Anti-Debug.....	3
3.Trouver un triplet valide.....	9
4.Conclusion.....	13

1.Introduction

Je propose ici une solution au crackme 6 de Thierry The One. Ce tutorial fait partie d'un projet initié par Gamera pour la NAS (powaaaa) et regroupant les 8 solutions des 8 crackme de TTO.

Dans la suite de ce tutorial, **toute valeur sera systématiquement exprimée en hexadécimal** sauf indication contraire.

2. Anti-Debug

a) Généralités

Allez on est parti ! On lance le crackme et on observe...3 edits, un pour le name, et 2 pour des serials.

Le crackme n'est pas packé, on peut donc le charger directement dans OllyDbg. Direction le bas du listing là où sont tous les JMP DWORD PTR... On pose un breakpoint sur GetWindowTextA (l'une des fonctions les plus courantes pour récupérer du texte) et on lance avec F9.

```
00401744  $-FF25 38204000 JMP DWORD PTR DS:[<&user32.EnumWindows>] user32.EnumWindows
0040174A  $-FF25 3C204000 JMP DWORD PTR DS:[<&user32.GetClassNameA>] user32.GetClassNameA
00401750  $-FF25 74204000 JMP DWORD PTR DS:[<&user32.GetDlgItem>] user32.GetDlgItem
00401756  $-FF25 44204000 JMP DWORD PTR DS:[<&user32.GetMessageA>] user32.GetMessageA
0040175C  $-FF25 48204000 JMP DWORD PTR DS:[<&user32.GetSystemMet>] user32.GetSystemMetrics
00401762  $-FF25 4C204000 JMP DWORD PTR DS:[<&user32.GetWindowTextA>] user32.GetWindowTextA
00401768  $-FF25 50204000 JMP DWORD PTR DS:[<&user32.KillTimer>] user32.KillTimer
0040176E  $-FF25 54204000 JMP DWORD PTR DS:[<&user32.MessageBoxA>] user32.MessageBoxA
00401774  $-FF25 58204000 JMP DWORD PTR DS:[<&user32.RegisterClassExA>] user32.RegisterClassExA
0040177A  $-FF25 5C204000 JMP DWORD PTR DS:[<&user32.SendMessageA>] user32.SendMessageA
00401780  $-FF25 60204000 JMP DWORD PTR DS:[<&user32.SetTimer>] user32.SetTimer
00401786  $-FF25 64204000 JMP DWORD PTR DS:[<&user32.SetWindowLongA>] user32.SetWindowLongA
0040178C  $-FF25 68204000 JMP DWORD PTR DS:[<&user32.ShowWindow>] user32.ShowWindow
00401792  $-FF25 6C204000 JMP DWORD PTR DS:[<&user32.TranslateMes>] user32.TranslateMessage
00401798  $-FF25 70204000 JMP DWORD PTR DS:[<&user32.UpdateWindow>] user32.UpdateWindow
0040179E  $-FF25 00204000 JMP DWORD PTR DS:[<&comctl32.InItCommon>] comctl32.InItCommonControls
004017A4  00 DB 00
004017A5  00 DB 00
```

La petite surprise c'est qu'au bout d'un temps très court Olly est tout simplement fermé ! TTO avait prévenu, il y a des anti-debug !

Bon ben on va tracer au fur et à mesure alors ! C'est ptête pas très réjouissant mais au moins on verra bien comment fonctionne le code. Reprenons depuis le début :

```
00401377  $ 6A 00 PUSH 0
00401379  . E8 90030000 CALL <JMP.&kernel32.GetModuleHandleA> GetModuleHandleA
0040137E  . A3 40304000 MOV DWORD PTR DS:[403040],EAX
00401383  . E8 16040000 CALL <JMP.&comctl32.InItCommonControls> InItCommonControls
00401388  . 6A 00 PUSH 0
0040138A  . 68 A5134000 PUSH tto_Crac.004013A5
0040138F  . 6A 00 PUSH 0
00401391  . 6A 65 PUSH 65
00401393  . FF35 40304000 PUSH DWORD PTR DS:[403040]
00401399  . E8 94030000 CALL <JMP.&user32.DialogBoxParamA> DialogBoxParamA
0040139E  . 6A 00 PUSH 0
004013A0  . E8 63030000 CALL <JMP.&kernel32.ExitProcess> ExitProcess
```

Première chose que l'on remarque l'appel à DialogBoxParamA qui indique que la fonction Callback du programme est située en 4013A5 (paramètre DlgProc). Il faut aussi savoir que cette fonction envoie un message WM_INITDIALOG à la Callback sitôt son travail terminé. Il est donc logique d'aller examiner la Callback et voir ce qu'elle fait lorsqu'elle traite un message WM_INITDIALOG.

Note : la fonction Callback d'un programme est chargée de traiter des messages fournis par Windows (par exemple si une touche à

été appuyée, si on demande la fermeture, si on redimensionne la fenêtre...). Chaque fenêtre peut posséder sa fonction Callback.

<pre> 004013A5 . 55 PUSH EBP 004013A6 . 8BEC MOV EBP,ESP 004013A8 . 8B45 0C MOV EAX,DWORD PTR SS:[EBP+C] 004013AB . 3D 10010000 CMP EAX,110 004013B0 . 75 0F85 8B000000 JNZ tto_Crac.00401441 004013B6 . 68 C8000000 PUSH 0C8 004013BB . FF75 08 PUSH DWORD PTR SS:[EBP+8] 004013BE . E8 8D030000 CALL <JMP.&user32.GetDlgItem> 004013C3 . A3 F0304000 MOV DWORD PTR DS:[4030F0],EAX 004013C8 . 68 00104000 PUSH tto_Crac.00401000 004013CD . 6A FC PUSH -4 004013CF . FF95 F0304000 PUSH DWORD PTR DS:[4030F0] 004013D5 . E8 AC030000 CALL <JMP.&user32.SetWindowLongA> 004013DA . A3 FC304000 MOV DWORD PTR DS:[4030FC],EAX 004013DF . 68 C9000000 PUSH 0C9 004013E4 . FF75 08 PUSH DWORD PTR SS:[EBP+8] 004013E7 . E8 64030000 CALL <JMP.&user32.GetDlgItem> 004013EC . A3 F4304000 MOV DWORD PTR DS:[4030F4],EAX 004013F1 . 68 CA000000 PUSH 0CA 004013F6 . FF75 08 PUSH DWORD PTR SS:[EBP+8] 004013F9 . E8 52030000 CALL <JMP.&user32.GetDlgItem> 004013FE . A3 F8304000 MOV DWORD PTR DS:[4030F8],EAX 00401403 . 6A 00 PUSH 0 00401405 . 68 E8030000 PUSH 3E8 0040140A . 6A 0D PUSH 0D 0040140C . FF75 08 PUSH DWORD PTR SS:[EBP+8] 0040140F . E8 6C030000 CALL <JMP.&user32.SetTimer> 00401414 . 68 DD304000 PUSH tto_Crac.004030DD 00401419 . 6A 00 PUSH 0 0040141B . 6A 00 PUSH 0 0040141D . 68 0D114000 PUSH tto_Crac.0040110D 00401422 . 6A 00 PUSH 0 00401424 . 68 A1304000 PUSH tto_Crac.004030A1 00401429 . E8 D4020000 CALL <JMP.&kernel32.CreateThread> 0040142E . A3 E1304000 MOV DWORD PTR DS:[4030E1],EAX 00401433 . B8 00000000 MOV EAX,0 00401438 . C9 LEAVE 00401439 . C2 1000 RETN 10 </pre>	<pre> WM_INITDIALOG [ControlID = C8 (200.) hWnd [GetDlgItem] [NewValue = 401000 Index = GWL_WNDPROC hWnd = NULL [SetWindowLongA]] [ControlID = C9 (201.) hWnd [GetDlgItem]] [ControlID = CA (202.) hWnd [GetDlgItem]] [Timerproc = NULL Timeout = 1000. ms TimerID = D (13.) hWnd [SetTimer]] [pthreadId = tto_Crac.004030DD CreationFlags = 0 pthreadParm = NULL ThreadFunction = tto_Crac.0040110D StackSize = 0 pSecurity = tto_Crac.004030A1 [CreateThread]]] </pre>
--	--

Chaque message possède une valeur particulière unique qui permet de le reconnaître et de faire des actions en conséquence. La valeur de WM_INITDIALOG est 110, le traitement approprié commence donc en 4013B6.

Regardons un peu. Les choses intéressantes sont les appels à SetWindowLongA, SetTimer et CreateThread. Nous allons les étudier successivement.

b) SetWindowLongA

Passons les appels à GetDlgItem qui n'ont guère d'intérêt et arrêtons nous sur l'appel à SetWindowLongA. Cette fonction sert à changer les paramètres d'une fenêtre, suivant le paramètre Index. Ici il s'agit de GWL_WNDPROC qui sert à modifier l'adresse de la Callback. Dans notre cas la nouvelle adresse est 401000 et contient du code intéressant :

00401000	. 55	PUSH EBP	
00401001	. 8BEC	MOV EBP,ESP	
00401003	. 817D 0C 01010	CMV DWORD PTR SS:[EBP+C],101	
0040100A	. 75 1E	JNZ SHORT tto_Crac.0040102A	WM_KEYUP
0040100C	. 833D 6C314000	CMV DWORD PTR DS:[40316C],20	Si ce n'est pas le bon message on
00401013	. 74 15	JE SHORT tto_Crac.0040102A	le refile a la premiere Callback
00401015	. 8B45 10	MOV EAX,DWORD PTR SS:[EBP+10]	
00401018	. 8B1D 6C314000	MOV EBX,DWORD PTR DS:[40316C]	Recupere le compteur de caractere
0040101E	. 8883 0C314000	MOV BYTE PTR DS:[EBX+40310C],AL	Sauvegarde du caractere
00401024	. FF05 6C314000	INC DWORD PTR DS:[40316C]	Incremente le compteur de caractere
0040102A	> FF75 14	PUSH DWORD PTR SS:[EBP+14]	lParam
0040102D	. FF75 10	PUSH DWORD PTR SS:[EBP+10]	wParam
00401030	. FF75 0C	PUSH DWORD PTR SS:[EBP+C]	Message
00401033	. FF35 F0304000	PUSH DWORD PTR DS:[4030F0]	hwnd = 000E04A6 (class='Edit',parent=000E04C4)
00401039	. FF35 FC304000	PUSH DWORD PTR DS:[4030FC]	PrevProc = user32.77D3B3C4
0040103F	. E8 DC060000	CALL <JMP.&user32.CallWindowProcA>	CallWindowProcA
00401044	. C9	LEAVE	
00401045	. C2 1000	RETN 10	

Cette fonction se charge d'intercepter uniquement les message WM_KEYUP qui indiquent qu'une touche vient d'être relachée, tout autre message étant renvoyé vers la première Callback. Une fois le message intercepté, on stocke le caractère de la touche relevée en 40310C et on incrémente le nombre de caractères traités en 40316C. Cette manière de traiter les caractères fait que malheureusement même un appui sur Shift est enregistré. Nous verrons aussi plus tard qu'il s'agit spécifiquement du traitement du name.

c) CreateThread

00401414	. 68 DD304000	PUSH tto_Crac.004030DD	pThreadId = tto_Crac.004030DD
00401419	. 6A 00	PUSH 0	CreationFlags = 0
0040141B	. 6A 00	PUSH 0	pThreadParam = NULL
0040141D	. 68 0D114000	PUSH tto_Crac.0040110D	ThreadFunction = tto_Crac.0040110D
00401422	. 6A 00	PUSH 0	StackSize = 0
00401424	. 68 A1304000	PUSH tto_Crac.004030A1	pSecurity = tto_Crac.004030A1
00401429	. E8 D4020000	CALL <JMP.&kernel32.CreateThread>	CreateThread

Il y a création d'un thread donc l'adresse de début est 40110D (paramètre ThreadFunction).

Note : un thread est une sorte de petit processus mais qui est lui interne au programme et partage son espace mémoire.

On tombe ici sur un début de programme courant : RegisterClassExA, CreateWindowExA, ShowWindow, UpdateWindow...et SetTimer ! C'est ce dernier qui va capter notre attention ! La fenetre associée est celle qui vient d'etre créée par CreateWindowExA et dont nous devons trouver la Callback (qui recevra donc les messages WM_TIMER).

Reflechissons....

On sait que RegisterClassExA prend en paramètre un pointeur sur une structure WNDCLASSEX qui est dans notre cas 4030AD. L'adresse de la Callback est le 3e DWORD de la structure WNDCLASSEX donc :

```

0040110D . 55          PUSH EBP
0040110E . 8BEC       MOV EBP,ESP
00401110 . 83C4 04    ADD ESP,-2C
00401113 . C705 AD304000 MOV DWORD PTR DS:[4030AD],30
0040111D . C705 B1304000 MOV DWORD PTR DS:[4030B1],2003
00401127 . C705 B5304000 MOV DWORD PTR DS:[4030B5],tto_Crac.004016B9
00401131 . C705 B9304000 MOV DWORD PTR DS:[4030B9],0
00401138 . C705 BD304000 MOV DWORD PTR DS:[4030BD],0
00401145 . FF35 40304000 PUSH DWORD PTR DS:[403040]
00401148 . 8F05 C1304000 POP DWORD PTR DS:[4030C1]
00401151 . 6A 06      PUSH 6
00401153 . 8F05 D1304000 POP DWORD PTR DS:[4030D1]
00401159 . C705 D1304000 MOV DWORD PTR DS:[4030D1],0
00401163 . C705 D5304000 MOV DWORD PTR DS:[4030D5],tto_Crac.00403027
0040116D . C705 C5304000 MOV DWORD PTR DS:[4030C5],0
00401177 . C705 C9304000 MOV DWORD PTR DS:[4030C9],0
00401181 . C705 D9304000 MOV DWORD PTR DS:[4030D9],0
00401188 . 68 AD304000 PUSH tto_Crac.004030AD
00401190 . E8 DF050000 CALL <JMP.&user32.RegisterClassExA>
00401195 . 6A 00      PUSH 0
00401197 . E8 C0050000 CALL <JMP.&user32.GetSystemMetrics>

```

On push l'adresse de la Callback

ASCII "Anti-Debugger ;)"

plWndClassEx = tto_Crac.004030AD
RegisterClassExA
Index = SMLCXSCREEN
GetSystemMetrics

4016B9 est bien le 3e DWORD de la structure ! Si c'est pas formidable ca de trouver l'adresse comme ca en clair sans avoir besoin de debugger !

N'oublions pas que nous avons un petit SetTimer sur le feu. Faudrait ptete aller voir en 4016B9 maintenant !

```

004016B9 . 55          PUSH EBP
004016BA . 8BEC       MOV EBP,ESP
004016BC . 817D 0C 13010 CMP DWORD PTR SS:[EBP+C],113
004016C3 . 75 11      JNZ SHORT tto_Crac.004016D6
004016C5 . FF05 9D304000 INC DWORD PTR DS:[40309D]
004016CB . B8 00000000 MOV EAX,0
004016D0 . C9        LEAVE
004016D1 . C2 1000    RETN 10
004016D4 . EB 10     JMP SHORT tto_Crac.004016E6
004016D6 . 837D 0C 10 CMP DWORD PTR SS:[EBP+C],10
004016DA . 75 0A     JNZ SHORT tto_Crac.004016E6
004016DC . 6A 0E     PUSH 0E
004016DE . FF75 08   PUSH DWORD PTR SS:[EBP+8]
004016E1 . E8 82000000 CALL <JMP.&user32.KillTimer>
004016E6 . FF75 14   PUSH DWORD PTR SS:[EBP+14]
004016E9 . FF75 10   PUSH DWORD PTR SS:[EBP+10]
004016EC . FF75 0C   PUSH DWORD PTR SS:[EBP+C]
004016EF . FF75 08   PUSH DWORD PTR SS:[EBP+8]
004016F2 . E8 35000000 CALL <JMP.&user32.DefWindowProcA>
004016F7 . C9        LEAVE
004016F8 . C2 1000    RETN 10

```

WM_TIMER

On incremente le compteur situe en 40309D

WM_CLOSE

TimerID = E (14.)
hWnd
KillTimer
lParam
wParam
Message
hWnd
DefWindowProcA

La fonction de traitement des messages n'est pas bien grande. Si on a le message WM_TIMER alors on incrémente le compteur situé en 40309D sinon si c'est un WM_CLOSE on tue le timer.

d) SetTimer

Arrêtons nous maintenant sur le SetTimer (euh rien à voir avec le précédent, c'est celui encore avant).

```

00401403 . 6A 00      PUSH 0
00401405 . 68 E0030000 PUSH 3E8
0040140A . 6A 00      PUSH 0D
0040140C . FF75 08   PUSH DWORD PTR SS:[EBP+8]
0040140F . E8 6C030000 CALL <JMP.&user32.SetTimer>

```

Timerproc = NULL
Timeout = 1000. ms
TimerID = D (13.)
hWnd
SetTimer

On sait que le timer va envoyer des messages WM_TIMER (de valeur 113) à la Callback toute les secondes. On va donc suivre la liste des messages traités.

Le premier est WM_INITDIALOG (cf 2a). Ce n'est pas ca donc on continue en 401441

```
00401441 > 3D 11010000 | CMP EAX,111 | WM_COMMAND
00401446 |.v0F85 8E010000 | JNZ tto_Crac.004015DA
```

Ici c'est WM_COMMAND (valeur 111), on continue en 4015DA.

```
004015DA |> 83F8 10 | CMP EAX,10 | WM_CLOSE
004015DD |.v75 24 | JNZ SHORT tto_Crac.00401603
```

Toujours pas ! Allons en 401603.

```
00401603 > 3D 13010000 | CMP EAX,113 | WM_TIMER
00401608 |.v0F85 99000000 | JNZ tto_Crac.004016A7 |
0040160E |.E8 D5FAFFFF | CALL tto_Crac.004010E8 | [<= 1er appel
00401613 |.50 | PUSH EAX | [lParam
00401614 |.68 48104000 | PUSH tto_Crac.00401048 | Callback = tto_Crac.00401048
00401619 |.E8 26010000 | CALL <JMP.&user32.EnumWindows> | EnumWindows
0040161E |.E8 C5FAFFFF | CALL tto_Crac.004010E8 | [<= 2e appel
00401623 |.58 | POP EAX |
00401624 |.5B | POP EBX |
00401625 |.FF35 9D304000 | PUSH DWORD PTR DS:[40309D] | 40309D contient une valeur etudiee + tard
0040162B |.3BC3 | CMP EAX,EBX |
0040162D |.v75 29 | JNZ SHORT tto_Crac.00401658 | Saut BadBoy
0040162F |.3D B0000000 | CMP EAX,0B0 |
00401634 |.v75 22 | JNZ SHORT tto_Crac.00401658 | Saut BadBoy
00401636 |.25 FF000000 | AND EAX,0FF |
0040163B |.3D B0000000 | CMP EAX,0B0 |
00401640 |.v75 16 | JNZ SHORT tto_Crac.00401658 | Saut BadBoy
00401642 |.5B | POP EBX |
00401643 |.A1 9D304000 | MOV EAX,DWORD PTR DS:[40309D] |
00401648 |.2BC3 | SUB EAX,EBX |
0040164A |.83F8 01 | CMP EAX,1 |
0040164D |.v7F 09 | JG SHORT tto_Crac.00401658 | Entre le debut de la routine et ici, 40309D
0040164F |.B8 00000000 | MOV EAX,0 | ne doit pas avoir augmente de plus que 1
00401654 |.C9 | LEAVE |
00401655 |.C2 1000 | RETN 10 |
00401658 > 6A 00 | PUSH 0 | [lParam = 0
0040165A |.6A 00 | PUSH 0 | wParam = 0
0040165C |.6A 10 | PUSH 10 | Message = WM_CLOSE
0040165E |.FF75 08 | PUSH DWORD PTR SS:[EBP+8] | hWnd
00401661 |.E8 14010000 | CALL <JMP.&user32.SendMessageA> | SendMessageA
00401666 |.B8 00000000 | MOV EAX,0 |
0040166B |.C9 | LEAVE |
0040166C |.C2 1000 | RETN 10 |
```

Ca y est ! On est tombé sur la routine qui traite les messages WM_TIMER.

Juste comme ca : vous remarquez peut-être que la valeur située entre 40309D ne doit pas avoir augmenté de plus de 1 entre le début et la fin de la routine. Nous verrons un peu plus loin de quoi il s'agit.

Ca commence avec un appel à 401048.

```
004010E8 |.$ 8D05 7B304000 | LEA EAX,DWORD PTR DS:[40307B] |
004010EE |.48 | DEC EAX |
004010EF |.BB FFFFFFFF | MOV EBX,-1 |
004010F4 |.B9 00000000 | MOV ECX,0 |
004010F9 > 40 | INC EAX |
004010FA |.43 | INC EBX |
004010FB |.83FB 1E | CMP EBX,1E |
004010FE |.v74 04 | JE SHORT tto_Crac.00401104 |
00401100 |.0208 | ADD CL,BYTE PTR DS:[EAX] |
00401102 |.^EB F5 | JMP SHORT tto_Crac.004010F9 |
00401104 > 58 | POP EAX |
00401105 |.51 | PUSH ECX |
00401106 |.50 | PUSH EAX |
00401107 |.E8 08060000 | CALL <JMP.&kernel32.IsDebuggerPresent> | IsDebuggerPresent
0040110C |.C3 | RETN
```

La première partie n'est pas très intéressante, elle ne fait que

quelques verifications d'intégrité. Par contre il faut faire attention à l'appel à IsDebuggerPresent (fonction déterminant si un programme est débuggé ou non). En effet si l'on ne fait rien, cela pourrait nous emmener tout droit sur un BadBoy. Le plus simple est d'utiliser un plugin de Olly permettant de cacher notre debugger.

Viens ensuite l'appel à EnumWindows. Comme son nom l'indique cette fonction énumère toutes les fenêtres une par une et les fait traiter par la fonction passée en paramètre (ici 401048).

<pre> 00401048 . 55 PUSH EBP 00401049 . 8BEC MOV EBP,ESP 0040104B . 83C4 80 ADD ESP,-80 0040104E . 837D 0C 00 CMP DWORD PTR SS:[EBP+C],0 00401052 > 75 61 JNZ SHORT tto_Crac.004010B5 00401054 . 68 80000000 PUSH 80 00401059 . 8D45 80 LEA EAX,DWORD PTR SS:[EBP-80] 0040105C . 50 PUSH EAX 0040105D . FF75 08 PUSH DWORD PTR SS:[EBP+8] 00401060 . E8 E5060000 CALL <JMP.&user32.GetClassNameA> 00401065 . 68 7B304000 PUSH tto_Crac.0040307B 0040106A . 8D45 80 LEA EAX,DWORD PTR SS:[EBP-80] 0040106D . 50 PUSH EAX 0040106E . E8 A7060000 CALL <JMP.&kernel32.lstrcpA> 00401073 . 83F8 00 CMP EAX,0 00401076 > 74 26 JE SHORT tto_Crac.0040109E 00401078 . 68 83304000 PUSH tto_Crac.00403083 0040107D . 8D45 80 LEA EAX,DWORD PTR SS:[EBP-80] 00401080 . 50 PUSH EAX 00401081 . E8 94060000 CALL <JMP.&kernel32.lstrcpA> 00401086 . 83F8 00 CMP EAX,0 00401089 > 74 13 JE SHORT tto_Crac.0040109E 0040108B . 68 8E304000 PUSH tto_Crac.0040308E 00401090 . 8D45 80 LEA EAX,DWORD PTR SS:[EBP-80] 00401093 . 50 PUSH EAX 00401094 . E8 81060000 CALL <JMP.&kernel32.lstrcpA> 00401099 . 83F8 00 CMP EAX,0 0040109C > 75 27 JNZ SHORT tto_Crac.004010C5 0040109E > 6A 00 PUSH 0 004010A0 . 6A 00 PUSH 0 004010A2 . 6A 10 PUSH 10 004010A4 . FF75 08 PUSH DWORD PTR SS:[EBP+8] 004010A7 . E8 CE060000 CALL <JMP.&user32.SendMessageA> 004010AC . B8 00000000 MOV EAX,0 004010B1 . C9 LEAVE 004010B2 . C2 0800 RETN 8 004010B5 > 6A 00 PUSH 0 004010B7 . E8 4C060000 CALL <JMP.&kernel32.ExitProcess> 004010BC . B8 00000000 MOV EAX,0 004010C1 . C9 LEAVE 004010C2 . C2 0800 RETN 8 004010C5 > 8B45 08 MOV EAX,DWORD PTR SS:[EBP+8] 004010C8 . C9 LEAVE 004010C9 . C2 0800 RETN 8 </pre>	<pre> Count = 80 (128.) Buffer hWnd GetClassNameA String2 = "OLLYDBG" String1 lstrcpA La fenetre correspond => BadBoy String2 = "OWL_Window" String1 lstrcpA La fenetre correspond => BadBoy String2 = "TidaWindow" String1 lstrcpA Saute si la fenetre ne correspond pas lParam = 0 wParam = 0 Message = WM_CLOSE hWnd SendMessageA ExitCode = 0 ExitProcess Ouf sauver ! </pre>
--	--

Ah bah ca tient mais alors ! On trouve des string qui correspondent comme par hasard à des debugger connus ! Il s'agit en fait des noms de classes de fenetres (fournies par GetClassNameA) de OllyDbg (OLLYDBG), WinDasm (OWL_Window) et IDA (TidaWindow). Il y a donc des comparaisons pour verifier qu'aucun de ces debugger n'est actuellement lancé, sinon quoi on envoi un WM_CLOSE au debugger suivit d'un ExitProcess pour notre programme. La solution est toute simple : nopper le JE après la comparaison avec OLLYDBG (si bien sur vous utilisez Olly).

3. Trouver un triplet valide

Tous les anti-debug ayant été repérés, nous allons maintenant pouvoir passer en phase de debugging. Petite précaution toutefois : ne pas oublier de nopper en 401076 pour éviter que notre debugger se fasse fermer.

```
00401065 | . 68 7B304000 | PUSH tto_Crac.0040307B
0040106A | . 8D45 80      | LEA EAX, DWORD PTR SS:[EBP-80]
0040106D | . 50          | PUSH EAX
0040106E | . E8 A7060000 | CALL <JMP.&kernel32.lstrcpA>
00401073 | . 83F8 00     | CMP EAX, 0
00401076 | . 90         | NOP
00401077 | . 90         | NOP
```

```
String2 = "OLLYDBG"
String1
lstrcpA
La fenetre correspond => BadBoy
```

Il ne reste qu'à poser un BP sur les appels à GetWindowTextA (visibles par click droit -> Search For -> All intermodular Calls) et lancer le programme par F9.

Voilà le crackme se lance, on complète les infos (naspower/123456/987654 pour moi) on valide et on break ici :

```
00401463 | . 6A 20      | PUSH 20
00401465 | . 68 2C314000 | PUSH tto_Crac.0040312C
0040146A | . FF35 F4304000 | PUSH DWORD PTR DS:[4030F4]
00401470 | . E8 E0020000 | CALL <JMP.&user32.GetWindowTextA>
00401475 | . 83F8 00     | CMP EAX, 0
00401478 | . 74 F84 21010000 | JE tto_Crac.0040159F
0040147E | . 6A 20      | PUSH 20
00401480 | . 68 4C314000 | PUSH tto_Crac.0040314C
00401485 | . FF35 F8304000 | PUSH DWORD PTR DS:[4030F8]
0040148B | . E8 D2020000 | CALL <JMP.&user32.GetWindowTextA>
00401490 | . 83F8 00     | CMP EAX, 0
00401493 | . 74 F84 06010000 | JE tto_Crac.0040159F
```

```
Count = 20 (32.)
Buffer = tto_Crac.0040312C
hwnd = 000604C4 (class='Edit',parent=001103AA)
GetWindowTextA

Count = 20 (32.)
Buffer = tto_Crac.0040314C
hwnd = 00080442 (class='Edit',parent=001103AA)
GetWindowTextA
```

En regardant de plus près on s'aperçoit que les 2 appels ne récupèrent ici que les serial. Qu'en est-il du name ?

On peut tracer sans problème jusqu'en 40157C (avant il n'y a que des controles d'intégrité sans intérêt).

```
0040157C | . 833D 6C314000 | CMP DWORD PTR DS:[40316C], 0
00401583 | . 7E 1A        | JLE SHORT tto_Crac.0040159F
00401585 | . 833D 6C314000 | CMP DWORD PTR DS:[40316C], 0A
0040158C | . 7D 11        | JGE SHORT tto_Crac.0040159F
0040158E | . E8 CBFCFFFF | CALL tto_Crac.0040125E
00401593 | . 83F8 00     | CMP EAX, 0
00401596 | . 75 07        | JNZ SHORT tto_Crac.0040159F
```

```
<= Call tres interessant
```

Si vous vous rappelez du début de ce tutorial vous saurez que 40316C contient le nombre de caractère du name. Ici il est donc clair que le serial doit obligatoirement faire strictement moins de A caractères (pour ceux dont ce ne serait pas le cas, il faut relancer sans oublier de nopper la ou il faut !). De plus en regardant notre name en 40310C on s'aperçoit que toutes les lettres ont été passées en majuscules (notez ce détail ça servira plus tard).

Maintenant allons voir le Call que j'ai marqué comme intéressant.

En effet c'est lui qui détermine si les informations entrées sont correctes. Le petit CMP EAX, 0 juste après signifie qu'il faut que le Call renvoie une valeur nulle dans EAX.

40125E :

Adresse	Opcode	Instruction	Commentaire
0040125E	33C0	XOR EAX,EAX	RAZ des registres
00401260	33DB	XOR EBX,EBX	
00401262	33C9	XOR ECX,ECX	
00401264	33D2	XOR EDX,EDX	
00401266	8A8C18 0C3140	MOV CL, BYTE PTR DS:[EAX+EBX+40310C]	Charge un caractere du name
0040126D	888A 70314000	MOV BYTE PTR DS:[EDX+403170], CL	
00401273	8A8C18 2C3140	MOV CL, BYTE PTR DS:[EAX+EBX+40312C]	Charge un caractere du serial 1
0040127A	888A 71314000	MOV BYTE PTR DS:[EDX+403171], CL	
00401280	8A8C18 4C3140	MOV CL, BYTE PTR DS:[EAX+EBX+40314C]	Charge un caractere du serial 2
00401287	888A 72314000	MOV BYTE PTR DS:[EDX+403172], CL	
0040128D	83C2 02	ADD EDX, 2	
00401290	40	INC EAX	
00401291	83F8 01	CMP EAX, 1	
00401294	74 DD	JE SHORT tto_Crac.00401273	
00401296	4A	DEC EDX	
00401297	83F8 02	CMP EAX, 2	
0040129A	74 E4	JE SHORT tto_Crac.00401280	
0040129C	33C0	XOR EAX,EAX	
0040129E	803D 70314000	CMP BYTE PTR DS:[403170], 0	Verifie si l'on a terminer
004012A5	75 03	JNZ SHORT tto_Crac.004012AA	
004012A7	33C0	XOR EAX,EAX	Si oui on retourne 0 dans EAX
004012A9	C3	RETN	
004012AA	68 70314000	PUSH tto_Crac.00403170	PUSH la chaine fabriquée
004012AF	E8 14000000	CALL tto_Crac.004012C8	Simple controle de longueur
004012B4	43	INC EBX	
004012B5	83F8 00	CMP EAX, 0	
004012B8	74 01	JE SHORT tto_Crac.004012BB	
004012BA	C3	RETN	
004012BB	E8 2B000000	CALL tto_Crac.004012EB	Veritable verification...
004012C0	33D2	XOR EDX,EDX	
004012C2	83F8 00	CMP EAX, 0	...qui doit retourner elle aussi 0
004012C5	74 9F	JE SHORT tto_Crac.00401266	
004012C7	C3	RETN	

Dans cette routine on fabrique une chaine de 6 caractères composée dans l'ordre comme ceci :

- 1 caractère du name d'index i
- 1 caractère du serial 1 d'index i
- 1 caractère du serial 2 d'index i
- 1 caractère du serial 1 d'index i+1
- 1 caractère du serial 2 d'index i+1
- 1 caractère du serial 2 d'index i+2

Le Call vers 4012C8 n'est pas intéressant, il sert juste ici à vérifier que cette chaine fait bien 6 caractères de long.

Bien entendu cette boucle de vérification sera répétée tant qu'il y aura des caractères dans le name le serial 1 aura donc au moins 1 caractère de plus que le name et le serial 2 en aura au moins 2 de plus.

Intéressons nous maintenant au Call de 4012EB.

Encore une fois un peu d'instructions sans intérêt avant le vrai commencement en 40132D

00401320	. 330B	XOR EBX,EBX	
0040132F	. 8A1D 71314000	MOV BL,BYTE PTR DS:[403171]	
00401335	. 2A1D 70314000	SUB BL,BYTE PTR DS:[403170]	
0040133B	. 80FB 01	CMP BL,1	
0040133E	> 75 26	JNZ SHORT tto_Crac.00401366	Char1 - Char0 = 1
00401340	. 330B	XOR EBX,EBX	
00401342	. 8A1D 72314000	MOV BL,BYTE PTR DS:[403172]	
00401348	. 321D 73314000	XOR BL,BYTE PTR DS:[403173]	
0040134E	. 80FB 2A	CMP BL,2A	
00401351	> 75 13	JNZ SHORT tto_Crac.00401366	Char2 xor Char3 = 2A
00401353	. 330B	XOR EBX,EBX	
00401355	. 8A1D 74314000	MOV BL,BYTE PTR DS:[403174]	
0040135B	. 221D 75314000	AND BL,BYTE PTR DS:[403175]	
00401361	. 80FB 62	CMP BL,62	
00401364	> 74 AD	JE SHORT tto_Crac.00401313	Char4 and Char5 = 62
00401366	> 5B	POP EBX	
00401367	. 5B	POP EBX	
00401368	. B8 01000000	MOV EAX,1	
0040136D	. C3	RETN	

Voilà nous avons enfin toutes les cartes dans nos mains !

Seulement ces 3 verifications devant être vérifiées pour toute chaine de caractères fabriquées, les choix vont être limités. Pour ma part j'ai pris la solution de facilité : je suppose que tous les caractères du serial 2 sont 62 ('b') (car 2 caractères successifs du serial 2 ANDé doivent donner 62). Je XOR 62 à 2A = 48 ('H') ce qui me donne tous les caractères du serial 1. Il ne reste qu'à soustraire 1 pour trouver les caractères du name qu'il faut repasser en minuscule.

On trouve donc le triplet gagnant (par exemple) : ggggg/HHHHHH/bbbbbbb

Mais tout est-il réellement fini ?

On clique un première fois rien ne se passe. Par contre on clique une deuxième fois et l'on a le droit au message « Bravo ». L'explication se trouve en 4015B0 peu après le Call intéressant de tout à l'heure :

00401585	. 833D 6C314000	CMP DWORD PTR DS:[40316C],0A	
0040158C	> 7D 11	JGE SHORT tto_Crac.0040159F	
0040158E	. E8 CBFCFFFF	CALL tto_Crac.0040125E	<= Call tres interessant
00401593	. 83F8 00	CMP EAX,0	
00401596	> 75 07	JNZ SHORT tto_Crac.0040159F	
00401598	. 68 6F164000	PUSH tto_Crac.0040166F	
0040159D	> EB 05	JMP SHORT tto_Crac.004015A4	
0040159F	> 68 8A164000	PUSH tto_Crac.0040168A	
004015A4	> 8F05 99304000	POP DWORD PTR DS:[403099]	
004015AA	. FF35 99304000	PUSH DWORD PTR DS:[403099]	
004015B0	. E8 17FBFFFF	CALL tto_Crac.004010CC	<= Decrypt
004015B5	. A1 9D304000	MOV EAX,DWORD PTR DS:[40309D]	
004015BA	. 5B	POP EBX	
004015BB	. 2BC3	SUB EAX,EBX	
004015BD	. 83F8 01	CMP EAX,1	
004015C0	> 0F8F 92000000	JG tto_Crac.00401658	
004015C6	. FF15 99304000	CALL DWORD PTR DS:[403099]	Appel de l'endroit decrypter
004015CC	. B8 00000000	MOV EAX,0	
004015D1	. C9	LEAVE	
004015D2	. C2 1000	RETN 10	

Une fois que nos informations ont été vérifiées et acceptée, un appel à une fonction de decryptage à lieu. Or l'endroit doit être decrypter 2 fois avant de révéler un appel à une MessageBox de félicitations :

0040166F	. C3	RETN	La premiere fois on tombe sur un RET
00401670	? 59	POP ECX	
00401671	. C159 89 99	ROR DWORD PTR DS:[ECX-77],99	Shift constant out of range 1..31
00401675	? 59	POP ECX	
00401676	? C1C3 89	ROL EBX,89	Shift constant out of range 1..31
00401679	. 99	CDQ	
0040167A	. 59	POP ECX	
0040167B	? 58	POP EAX	
0040167C	. 8E95 89995941	MOV SS,WORD PTR SS:[EBP+41599989]	Modification of segment register
00401682	41	DB 41	CHAR 'A'
00401683	59	DB 59	CHAR 'Y'
00401684	59	DB 59	CHAR 'Y'
00401685	59	DB 59	CHAR 'Y'
00401686	22	DB 22	CHAR ''
00401687	1B	DB 1B	
00401688	69	DB 69	CHAR 'i'
00401689	. 59	DB 59	CHAR 'Y'
0040168A	. C3	RETN	

0040166F	. 6A 00	PUSH 0	La premiere fois on tombe sur un RET
00401671	. 68 00304000	PUSH tto_Crac.00403000	ASCII "[lvl 2.5] CrackMe6 par Thierry The One"
00401676	. 68 6A304000	PUSH tto_Crac.0040306A	ASCII "Bravo"
0040167B	? FF35 3C304000	PUSH DWORD PTR DS:[40303C]	
00401681	? E8 E8000000	CALL <JMP.&user32.MessageBoxA>	
00401686	C9	DB C9	
00401687	C2	DB C2	
00401688	10	DB 10	
00401689	. 00	DB 00	
0040168A	. C3	RETN	

Et voilà !

4. Conclusion

Ainsi s'achève l'étude de l'un des crackme que j'ai eu le plus de plaisir à étudier. En effet les différentes interconnexions des parties du code apportent un petit plus à ce crackme.

Je tiens donc à féliciter Thierry The One pour ce crackme !
Et en espérant qu'il y en ai d'autres.

Encore une fois je tiens aussi à remercier Gamera pour son projet !

That's all folks !

26/09/06
mastermatt29^NAS